A Quality of Service Catalog for Software Components

Girish J. Brahnmath[1] Rajeev R. Raje[1] Andrew M. Olson[1] Mikhail Auguston[2] Barrett R. Bryant[3] Carol C. Burt[3]

Abstract

Component-based Software Development is being recognized as the direction in which the software industry is headed. With the proliferation of Commercial Off The Shelf (COTS) Components, this trend will continue to emerge as a preferred technique for developing distributed software systems encompassing heterogeneous components. In order for this approach to result in software systems with a predictable quality, the COTS components utilized should in turn offer a guaranteed level of quality. This calls for an objective paradigm for quantifying the quality of service of COTS components. A Quality of Service (QoS) catalog, proposed here, for software components is a first step in quantifying the quality attributes. This catalog is a critical component of the UniFrame project, which targets at unifying the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques.

Keywords: Quality of Service, non-functional attributes, QoS catalog, Component-based development.

1. Introduction:

Component-based software development uses appropriate off the shelf software components to create software systems. The notion of assembling complete systems out of prefabricated parts is prevalent in many branches of science and engineering such as manufacturing. This leads to the creation of prompt and economical products. This is possible because of the existence of standardized components that meet a manufacturer's functional and non-functional (quality) requirements. Also, the task of the manufacturer is made much easier because of the presence of standardized component catalogs outlining their functional and non-functional attributes.

At present, a software developer who uses the component-based approach cannot enjoy the same luxury. This is mainly because a majority of Commercial Off The Shelf (COTS) components are specified only with functional attributes in their interfaces. Typically, no concrete notion of quality is associated with components. Hence, the system developer has no means to objectively compare the performance characteristics of multiple components with the same functionality. This tends to restrict the developer's options when trying to select a component with a given functionality during the software development process. Thus, there is a need for a framework that would allow objective measurements of a component's Quality of Service (QoS) attributes. The creation of a Quality of Service catalog for software components would be the first step in this direction. Such a catalog should contain detailed descriptions about QoS attributes of software components along with the appropriate metrics, evaluation methodologies and the interrelationships with other attributes.

 As a part of the UniFrame project [1], we are creating a Quality of Service-based framework for distributed heterogeneous software components. It is expected that this framework would initiate a standardization process in the component-based software development community. This would prove to be beneficial to the COTS component developer (producer) and the system developer (consumer). It would enable the component developer to advertise the quality of his components by using the QoS metrics, and allow the system developer to verify and validate the claims of the component developer.

The rest of the paper is organized as follows. The next section contains a discussion about work related to QoS in other domains like networking and in the domain of software. In section 3, the QoS framework is described in detail, along with a brief description of the UniFrame project. In section 4, as an application of the QoS framework, a detailed case study is presented from the domain of banking. An outline of our future plans is presented in section 5. Finally, we conclude in section 6.

2. Related Work:

The notion of QoS has been largely associated with the field of networking. A number of architectures have been proposed for QoS guarantees for distributed multimedia systems. In [2], a quality of service architecture (QoS-A) to

[1]Department of Computer and Information Science, Indiana University Purdue University Indianapolis, {gbrahnma, rraje, aolson}@cs.iupui.edu; [2]Computer Science Department, Naval Post Graduate School (on leave from New Mexico State University) auguston@cs.nps.navy.mil; [3]Department of Computer and Information Sciences, The University of Alabama at Birmingham, {bryant, cburt}@cis.uab.edu.

Report Documentation Page

1. REPORT DATE **APR 2002**	2. REPORT TYPE	3. DATES COVERED **00-00-2002 to 00-00-2002**
4. TITLE AND SUBTITLE **A Quality of Service Catalog for Software Components**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Indiana University/Purdue University,Department of Computer and Information Sciences,Indianapolis,IN,46202**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
pp: 513-520, The Proceedings of the Southeastern Software Engineering Conference, Huntsville, AL, April 2002

14. ABSTRACT
Component-based Software Development is being recognized as the direction in which the software industry is headed. With the proliferation of Commercial Off The Shelf (COTS) Components, this trend will continue to emerge as a preferred technique for developing distributed software systems encompassing heterogeneous components. In order for this approach to result in software systems with a predictable quality, the COTS components utilized should in turn offer a guaranteed level of quality. This calls for an objective paradigm for quantifying the quality of service of COTS components. A Quality of Service (QoS) catalog, proposed here, for software components is a first step in quantifying the quality attributes. This catalog is a critical component of the UniFrame project, which targets at unifying the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT **Same as Report (SAR)**	18. NUMBER OF PAGES **8**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

specify and achieve the necessary performance properties of continuous media applications over asynchronous transfer mode (ATM) networks is proposed. In QoS-A, instead of considering the QoS in the end-system and the network separately, a new integrated approach, which incorporates QoS interfaces, control, and management mechanisms across all architectural layers, is used. This architecture is based on the notions of flow, service contract and flow management. A service contract makes it possible to formalize the QoS requirements of the user and the potential degree of service commitment of the service provider. It also enables the specification of the network resource requirements and the necessary actions to be taken in case of a service contract violation. Flow management is utilized to monitor and maintain the QoS specified in the service contract.

The Quality Objects (QuO) framework [3] provides QoS to distributed software applications composed of objects. QuO is intended to bridge the gap between the socket-level QoS and the distributed object level QoS. This work mainly emphasizes on specification, measurement, control and adaptation to changes in quality of service. QuO extends the CORBA functional IDL with a QoS description language (QDL). QDL is a suite of quality description languages for describing QoS contracts between clients and objects, the system resources and mechanisms for measuring and providing QoS and adaptive behavior on the client and object side. It utilizes the Aspect Oriented Programming paradigm [4], which provides support for incorporating the non-functional properties of components separately from the functional properties.

QoS Modeling Language (QML) is a QoS specification Language proposed in [5]. QML is an extension of UML. It is a general purpose QoS specification language capable of describing different QoS attributes in any application domain. If offers three main abstraction mechanisms for QoS specification: contract type, contract and profile. A contract type represents a specific QoS attribute like: reliability or performance and it defines dimensions that can be used to characterize a particular QoS attribute. A contract is defined as an instance of a contract type and it represents a particular QoS specification. Profiles are used to associate contracts with interface entities such as operations, operation arguments and operation results. Here, the QoS specifications are syntactically separate from interface definitions, allowing different implementations of the same service interface to have different QoS characteristics. Thus a service specification may comprise of a functional interface and one or more QoS specifications.

The following features of the UniFrame approach, for QoS, distinguish it from other related efforts:

1. A creation of a QoS Catalog for software components containing detailed descriptions about QoS attributes of software components including the metrics, evaluation methodologies and the interrelationships with other attributes.
2. An integration of QoS at the individual component and distributed system levels.
3. A formal specification, based on Two-Level Grammars (TLG) [6], of the QoS attributes of each component.
4. The validation and assurance of QoS, based on the concept of event grammars [7].
5. An investigation of the effects of component composition on QoS; involving the estimation of the QoS of an ensemble of software components given the QoS of individual components.
6. A QoS-centric iterative component-based software development process, to ensure that the end-product matches both the functional and QoS specifications.

In this paper, we have addressed only the first two features. The details of the other features are discussed in [1].

3. QoS Framework for Software components:
3.1 UniFrame Project:
Our work on the QoS framework is part of the Unified Meta Component Model Framework (UniFrame) project. The UniFrame research attempts to unify the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques. This research targets not only the dynamic assembly of distributed software systems from components built using different component models, but also the necessary instrumentation to enable QoS features of the component and the ensemble of components to be measured and validated. The core parts of UniFrame project are: components, service and service guarantees and infrastructure.

Component: In UniFrame, components are autonomous entities, whose implementations are non-uniform, i.e.; each component adheres to a distributed-component model but there is no notion of a unified implementation framework. Each component has a state, an identity, a behavior, a well-defined interface and a private implementation.

Service and Service Guarantees: A service offered by a component could be an intensive computational effort or an access to underlying resources. In a DCS, it is natural to expect several choices for obtaining a specific service. Thus, each component must be able to specify the quality of service (QoS) offered. The QoS is an indication given

by a component, on behalf of its owner, about its confidence to carry out the required services. The QoS offered by each component depends upon the computation it performs, the algorithm used, its expected computational effort, required resources, the motivation of the developer, and the dynamics of supply and demand.

Infrastructure: The headhunter and Internet Component Broker are responsible for allowing a seamless integration of different component models and sustaining cooperation among heterogeneous components. The tasks of headhunters are to detect the presence of new components in the search space, register their functionalities, and attempt matchmaking between service producers and consumers. It attempts at discovering components and registering them. Headhunters may cooperate with each other in order to serve a large number of components. The Internet Component Broker (ICB) acts as a translator between heterogeneous components. Adapter components register with ICB and indicate their specializations (which component models they can bridge efficiently). During a request from a seeker, the headhunter component not only searches for a provider, but also supplies the necessary details of an ICB.

Automated System Generation: In general, different developers will provide on the Internet a variety of possibly heterogeneous components oriented towards a specific problem domain. Once all the components necessary for implementing a specified distributed system are available and specific problem is formulated, then the task is to assemble them into a solution. UniFrame takes a pragmatic approach, based on Generative Programming [8,9], to component-based programming. It is assumed that the generation environment will be built around a generative domain-specific model (GDM) supporting component-based system assembly.

Further details about the UniFrame project can found in [1] [10] and [11].

3.2 Objectives of QoS Framework:

The QoS framework is a critical part of the UniFrame approach. The objectives of the QoS Framework are:

a) Identification of QoS attributes: A software component may be used in many different domains. Every domain has its own constraints with respect to the QoS attributes of software components. Hence, it is necessary to prepare a comprehensive compilation of different QoS attributes for many domains in which a software component may be used. Such a compilation would act as a checklist for any component developer/user interested in identifying the QoS attributes of interest.

b) Classification of QoS attributes based on:

 i. *Domain of usage:* Such a classification would enable a component user to identify the attributes that are relevant to his/her domain.

 ii. *Static / Dynamic behavior:* Such a classification would be helpful to determine whether a value of a QoS attribute is constant or varies according to the environment. This would in turn help in determining whether the value of a QoS attribute can be improved by changes to the operating environment.

 iii. *Nature of the attribute:* The QoS attributes identified are classified according to their characteristics into: *Time-related attributes (end-to-end-delay, freshness), Importance–related attributes (priority, precedence), Performance-related attributes (throughput, capacity), Integrity-related attributes (accuracy), Safety-related attributes (security) and Auxiliary attributes (portability, maintainability).*

 iv. *Composability of the attributes:* This kind of classification is important when different components are integrated to form a software system. It indicates whether the value of a given QoS attribute can be used in computing the value of the corresponding QoS attribute of the resultant system. Some of the QoS attributes are inherently non-composable, for example, parallelism constraints, priority, ordering constraints, etc. Hence, this classification would be valuable during the system integration phase.

c) Identification of metrics for QoS attributes: QoS metrics are the units for measuring the QoS attributes of a software component. Quantification of the QoS attributes of software components is one of the important goals of the proposed QoS framework. Hence, there is a need for standardized metrics to compare the QoS attributes of different software components. This would help to ensure uniformity in the expression of the QoS attributes.

d) Creation of a QoS catalog for Software Components: The QoS Catalog would act as a comprehensive source of information regarding the quality of software components. It would contain detailed descriptions about QoS attributes of software components including the metrics, evaluation methodologies and the interrelationships among the QoS attributes.

e) Creation of a QoS interface for a component with different levels of details: One of the primary objectives of the QoS framework is to make the QoS attributes an integral part of a software component. The QoS interface is aimed at achieving this objective. The QoS interface would contain the values for QoS attributes of a software component.

For the sake of brevity, here, only the concepts of QoS parameters and the QoS catalog are discussed.

3.3 Catalog of QoS Parameters:

The QoS Catalog for Software components would prove to be a valuable tool for:

i. The component developer by: a) acting as a reference manual for incorporating QoS attributes into the components being developed, b) allowing him to enhance the performance of his component in an iterative fashion by being able to quantify their QoS attributes, and c) enabling him to advertise the Quality of his components using the QoS metrics.

ii. The system developer by: a) enabling him to specify the QoS requirements of the components that are incorporated into his system, b) allowing him to verify and validate the claims of the component developer, c) allowing him to make objective comparisons of QoS of components having the same functionality, and d) empowering him with the means to choose the best suited components for his system.

At present the following QoS parameters have been selected for inclusion in the catalog. More parameters will be included as they are identified.

1. *Dependability:* It is a measure of confidence that the component is free from errors.
2. *Security:* It is a measure of the ability of the component to resist an intrusion.
3. *Adaptability:* It is a measure of the ability of the component to tolerate changes in resources and user requirements.
4. *Maintainability:* It is a measure of the ease with which a software system can be maintained.
5. *Portability:* It is a measure of the ease with which a component can be migrated to a new environment.
6. *Throughput:* It indicates the efficiency or speed of a component.
7. *Capacity:* It indicates the maximum number of concurrent requests a component can serve.
8. *Turn-around Time:* It is a measure of the time taken by the component to return the result.
9. *Parallelism Constraints:* It indicates whether a component can support synchronous or asynchronous invocations.
10. *Availability:* It indicates the duration when a component is available to offer a particular service.
11. *Ordering Constraints:* It indicates the order of returned results and its significance.
12. *Evolvability:* It indicates how easily a component can evolve over a span of time.
13. *Result:* Indicates the quality of the results returned.
14. *Achievability:* It indicates whether the component can provide a higher degree of service than promised.
15. *Priority:* It indicates if a component is capable of providing prioritized service.
16. *Presentation:* It indicates the quality of presentation of the results returned by the component.

Detailed sample descriptions of two of the above-mentioned QoS parameters, Dependability and Turn-around Time, are given below:

Name:	DEPENDABILITY
Intent:	It is a measure of confidence that the component is free from errors.
Description:	It is defined as the probability that the component is defect free.
Motivation:	1. It allows an evaluation of degree of Dependability of a given component.
	2. It allows a comparison of Dependability of different components.
	3. It allows for modifications to a component to increase its Dependability.
Applicability:	This parameter can be used in any system, which requires its components to offer a specific level of dependability. Using this parameter, the Dependability of a given component can be calculated before being incorporated into the system.
Model Used:	Dependability model by J. Voas and J. Payne [12].
Metrics used:	Testability Score, Dependability Score.
Influencing Factors:	1. Degree of testing.
	2. Fault hiding ability of the code.
	3. The likelihood that a statement in a component is executed.
	4. The likelihood that a mutated statement will infect the component's state.
	5. The likelihood that a corrupted state will propagate and cause the component output to be mutated.
Evaluation Procedure:	1. Perform Execution Analysis on the component.
	2. Perform Propagation Analysis on the component.
	3. Calculate the Testability value of the component.

Evaluation Formulae:	4. Calculate the Dependability Score of the component. T = E * P. *T: Testability Score (a prediction of the likelihood that a particular statement in a component will hide a defect during testing).* *E: Execution Estimate (the likelihood of executing a given fault).* *P: Propagation Estimate (the conditional probability of the corrupted data state corrupting the software's output after the state gets infected).* $D = 1 - (1-T)^N$. *D: Dependability Score.* *N: Number of successful tests.*
Result Type:	Floating Point Value between [0,1].
Static / Dynamic:	Static.
Composable / Non-Composable:	Composable.
Consequence:	1. Greater amounts of testing and greater Testability scores result in greater Dependability. 2. Lesser amount of testing is required to provide a fixed dependability score for higher Testability Scores.
Related Parameters:	Availability, Error Rate, Stability.
Domain of Usage:	Domain Independent.
Error Situation:	Low dependability results in: 1. Unreliable component behavior. 2. Improper execution / termination. 3. Erroneous results.
Aliases:	Maturity, Fault Hiding Ability, Degree of Testing.

Name:	**Turn-around Time**

Intent:	It is a measure of the time taken by the component to return the result.
Description:	It is defined as the time interval between the instant the component receives a request until the final result is generated.
Motivation:	1. It indicates the delay involved in getting results from a component. 2. It is one of the measures of the performance offered by a component.
Applicability:	This attribute can be used in any system, which specifies bounds on the response times of its components.
Model Used:	Empirical approach.
Metrics Used:	Mean Turn-around Time.
Influencing Factors:	1. Implementation (algorithm used, multi-thread mechanism etc). 2. Speed of the CPU. 3. Available memory. 4. Load on the system. 5. Operating System's access policy for resources like: CPU, I/O, memory, etc.
Evaluation Procedure:	1. Record the time instant at which the request is received. 2. Record the time instant at which the final result is produced. 3. Repeat steps 1 and 2 for 'n' representative requests. 4. Calculate the Mean Turn-around Time.
Evaluation Formulae:	$MTAT = [\sum_{i=1}^{n} (t2-t1)] / n$. MTAT: Mean Turn-around Time. t1: time instant at which the request is received. t2: time instant at which the final result is produced. n: number of representative requests.
Result Type:	Floating Point Value in milliseconds.
Static / Dynamic:	Dynamic.
Composable / Non-Composable	Composable.

Consequence:	Lower the time interval between the instant the request is received and the response is generated, lower the Mean Turn-around Time.
Related Parameters:	Throughput, Capacity.
Domain of Usage:	Domain Independent.
Error Situation:	A high value of Internal Response Time results in:
	1. Longer delays in producing the result.
	2. Higher round trip time.
Aliases:	Latency, Delay.

4. Case Study:

Let us assume that a private bank is trying to build a software system to automate its day-to-day operations. The bank has decided to utilize a Client-server Distributed computing model .The bank has also chosen to assemble the system using COTS software components instead of building the system from scratch.

The In-house software development team in the bank has come out with the following simple design for the system:
- The system consists of two categories of components: AccountServer and AccountClient.
- There will be two instances of the AccountServer and one instance of the AccountClient.
- The two AccountServers are of type javaAccountServer, adhering to the java-RMI model and corbaAccountServer, adhering to the CORBA model.
- The components should offer the following functionality: Deposit, Withdraw and Balance check

The system development team now needs three different components meeting the above functionality requirements. However, the bank also expects the components to satisfy certain QoS requirements. These are listed below:
- Dependability: The components will be an integral part of the bank and be responsible for keeping track of all transactions within the bank. Hence the component should offer some guarantees regarding error free operation.
- Turn-around Time: The transactions within the banking system have time restrictions imposed on them. Hence, they have to produce results within a specified time frame. This requires that the components satisfy Turn-around time requirements.

The partial UniFrame descriptions of these components are presented below:

JavaAccountServer:	**CorbaAccountServer:**
Informal Description: Provides an account management service. Supports three functions: javaDeposit(), javaWithdraw() and javaBalance().	Informal Description: Provides an account management service. Supports three functions: corbaDeposit(), corbaWithdraw() and corbaBalance().
1. Computational Attributes:	1. Computational Attributes:
a) Inherent Attributes:	a) Inherent Attributes:
a.1 id: intrepid.cs.iupui.edu/jServer	a.1 id: jovis.cs.iupui.edu/coServer
b) Functional Attributes:	b) Functional Attributes:
b.1 Acts as an account server	b.1 Acts as an account server
b.2 Algorithm: simple addition/subtraction	b.2 Algorithm: simple addition/subtraction
b.3 Complexity: O(1)	b.3 Complexity: O(1)
b.4 Syntactic Contract:	b.4 Syntactic Contract:
void javaDeposit(float ip);	void corbaDeposit(float ip);
void javaWithdraw(float ip) throws	void corbaWithdraw(float ip) throws
overDrawException;	overDrawException;
float javaBalance();	float corbaBalance();
b.5 Technology: Java-RMI	b.5 Technology: Java-CORBA
....	....
2. Cooperation Attributes:	2. Cooperation Attributes:
2.1) Pre-processing Collaborators: AccountClient	2.1) Pre-processing Collaborators: AccountClient
3. Auxiliary Attributes:	3. Auxiliary Attributes:
....	....
4. QoS Metrics:	4. QoS Metrics:
Dependability = 0.98	Dependability = 0.99
Turn-around Time: MTAT=70	Turn-around Time: MTAT=80

JavaAccountClient:

Informal Description: Requests account services from an appropriate server and interacts with the user; implemented as a web-based applet. Supports functions: depositMoney(), withdrawMoney() and checkBalance().

1. Computational Attributes:	2. Cooperation Attributes:
a) Inherent Attributes:	2.1) Post-processing Collaborators: AccountServer
a.1 id: galileo.cs.iupui.edu/aClient	
	3. Auxiliary Attributes:
b) Functional Attributes:	....
b.1 accepts user queries and presents the results	
using a GUI	4. QoS Metrics:
b.2 Algorithm: Java Foundation Classes (JFC)	Dependability = 0.99
b.3 Complexity: O(1)	Turn-around Time: MTAT = 90
b.4 Syntactic Contract	
void depositMoney(float ip);	
void withdrawMoney(float ip);	
float checkBalance();	
b.5 Technology: Java Applet	
....	

Query and Returned Results:

A sample query for the above example can be informally stated as: Create an account management system that has: Dependability > 0.97 and Turn-around Time: MTAT < 100. From the query and the available knowledge in the GDM associated with the account management systems, a formal specification of the desired system will be formulated for a headhunter in UniFrame. In response, the headhunter will discover the following choices:

1. Java-Java System: a) javaAccountClient -- Dependability = 0.99, Turn-around Time: MTAT = 90, Java Applet Technology b) javaAccountServer -- Dependability = 0.98, Turn-around Time: MTAT = 70, Java-RMI technology c) Infrastructure Needed -- JVM and Appletviewer.

2. Java-CORBA System: a) javaAccountClient -- Dependability = 0.99,Turn-around Time: MTAT= 90, Java Applet Technology b) corbaAccountServer – Dependability = 0.99,Turn-around Time: MTAT= 80, Java-RMI technology c) Infrastructure Needed -- JVM, Appletviewer, ORB, Java-CORBA bridge.

QoS of the assembled system:

Each component has two QoS parameters: 1) static – dependability and 2) dynamic - Turn-around Time. The desired QoS of the assembled system includes these parameters as well. For this reason the GDM will contain a rule that will compute the value of the static parameter for the assembled system. In this example, the dependability for the assembled system is calculated using the following formula: $(1.0 - ((1.0 - D_1) + (1.0 - D_2)))$, Where, D_1 and D_2 are the dependability values of the constituent components, yielding a value of 0.97 for the Java-Java System and a value of 0.98 for the Java-CORBA System.

For the dynamic parameter, the generator will provide the necessary instrumentation for taking the clock and calculating the Turn-around Time at run-time. The knowledge about metrics for the QoS parameter ` Turn-around Time' is represented in terms of Duration attribute for events of the type method-call, and the generic computation over the event trace that takes the clock and sums up those durations yielding a measured Turn-around Time for the accounting system.

One of the two example systems, mentioned in the query, will be implemented with the code for carrying out event trace computations according to user-supplied test cases. These test cases will be executed to verify that the accounting system satisfies the QoS specified in the query. If the system is not verified, it is discarded. This verification process is carried out for each of the generated accounting systems (two in the above example). Then, the one with the best QoS is chosen.

5. Future Plans:

Incorporation of the above-mentioned QoS parameters into the component interface is our next step. This would involve the creation of a QoS interface of the component along the lines of a functional (or syntactical) interface of a

component. This QoS interface would include all the necessary information about those QoS parameters that are selected by the component developer for inclusion in a given component. This would be followed by a formal specification of these QoS parameters and a mechanism for ensuring them at the individual component level and at the system level. The issue of Quality of Service of an ensemble of software components, i.e., a software system built out of components would also be addressed. This would involve the issues of component composition and composability of QoS Parameters.

6. Conclusion:

This paper has presented a QoS framework for software components, which is a part of the UniFrame project [1]. The objectives of the QoS framework include: a) the creation of a QoS catalog designed to quantify the QoS attributes of software components, b) incorporation of QoS attributes into the component interface, c) a formal specification of these attributes, d) a mechanism for ensuring these attributes at individual component level and at the system level, and e) a procedure to estimate the QoS of an ensemble of software components. Due to the space restrictions, only the concepts of QoS parameters and QoS catalog are presented here. The QoS framework would enable the component developer to advertise the quality of his components by using the QoS metrics, and allow the system developer to verify and validate the claims of the component developer. Although a simple case study is provided in this paper, the principles of the proposed approach are general enough to be applied to any larger applications.

7. References:

1) R. Raje, M. Auguston, B. Bryant, A. Olson, C. Burt. *A Quality of Service – based framework for creating distributed heterogeneous software components*, Technical Report, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2001.

2) A. Campbell. *A Quality of Service Architecture* –Ph.D. Thesis, Computing Department, Lancaster University, 1996.

3) BBN Corporation, Quality Objects Project, URL: *http://www.dist-systems.bbn.com/tech/QuO,* 2001.

4) Communications of ACM special issue on Aspect Oriented Programming, vol.44, No 10, October 2001.

5) S. Frolund, J. Koistinen. *Quality of Service specification in distributed object systems*, Distributed System Engineering Journal, Vol.5, No. 4, December, 1998

6) A. Van Wijngaarden. *Orthogonal Design and Description of a formal Language*. Technical Report, Mathematisch Centrum, Amsterdam, 1965.

7) M. Auguston. Program *Behavior Model Based on Event Grammar and it's Application for Debugging Automation*. In Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, 1995.

8) Batory, D. and Chen, G. and Robertson, E. and Wang, T. *Design Wizards and Visual Programming Environments for Gen Voca Generators*. IEEE Transactions on Software Engineering, pages 441-452, 2000.

9) Czarski, K., and Eisenecker, U.W. *Generative Programming: Methods, Tools, and Applications*. Addison - Wesley, 2000.

10) R. Raje. *"UMM: Unified Meta-object Model for Open Distributed Systems"*, Proceedings of the fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing pages 454-465 (ICA3PP' 2000).

11) R. Raje, M. Auguston, B. Bryant, A. Olson, C. Burt, *"A Unified Approach for the Integration of Distributed Heterogeneous Software Components"*, Proceedings of the 2001 Monterey Workshop (Sponsored by DARPA, ONR, ARO and AFOSR), Monterey, California, 2001.

12) J. Voas, J. Payne, *Dependability Certification of Software Components*, Journal of Systems and Software, NO. 52, pp. 165-172, 2000.